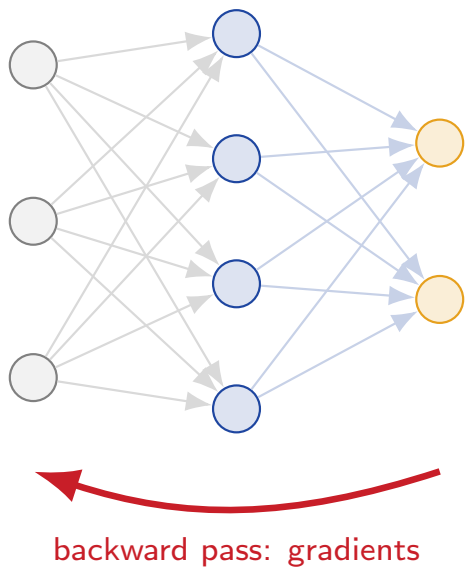
Training Neural Networks

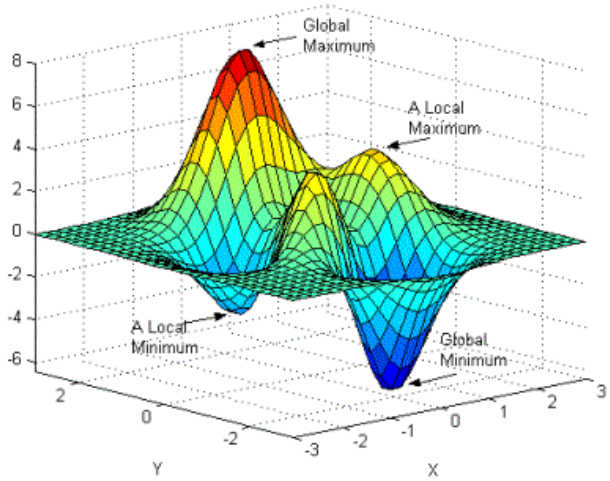
Gradient descent, computational graphs, and backpropagation

This deck embeds the LMU *Introduction to Deep Learning* training chunks in full (basic training → computational graphs → backpropagation → hardware & software), with our own notes sprinkled in.



Deep Learning

Basic Training



Learning goals

- Empirical risk minimization
- Gradient descent
- Stochastic gradient descent

TRAINING NEURAL NETWORKS

- In ML we use **empirical risk minimization** to minimize prediction losses over the training data

$$\mathcal{R}_{\text{emp}} = \frac{1}{n} \sum_{i=1}^{n} L\left(y^{(i)}, f\left(\mathbf{x}^{(i)} \mid \boldsymbol{\theta}\right)\right)$$

- In DL, $\boldsymbol{\theta}$ represents the weights (and biases) of the NN.
- Often, L2 in regression:

$$L\left(y, f(\mathbf{x})\right) = \frac{1}{2}(y - f(\mathbf{x}))^2$$

- or cross-entropy for binary classification

$$L\left(y, f(\mathbf{x})\right) = -(y \log f(\mathbf{x}) + (1 - y) \log(1 - f(\mathbf{x})))$$

- ERM can be implemented by **gradient descent**.

GRADIENT DESCENT

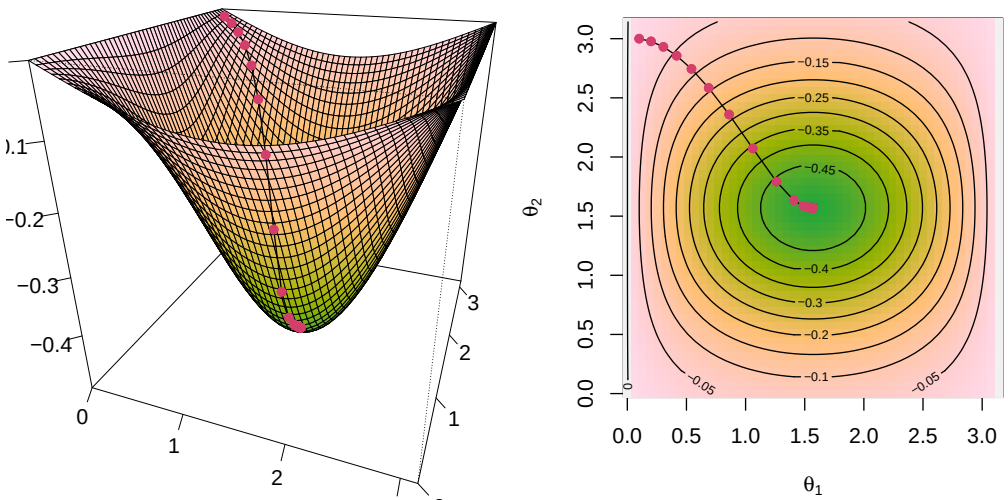
- Neg. risk gradient points in the direction of the **steepest descent**

$$-\mathbf{g} = -\nabla \mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) = -\left(\frac{\partial \mathcal{R}_{\text{emp}}}{\partial \theta_1}, \ldots, \frac{\partial \mathcal{R}_{\text{emp}}}{\partial \theta_d} \right)^\top$$

- "Standing" at a point $\boldsymbol{\theta}^{[t]}$, we locally improve by:

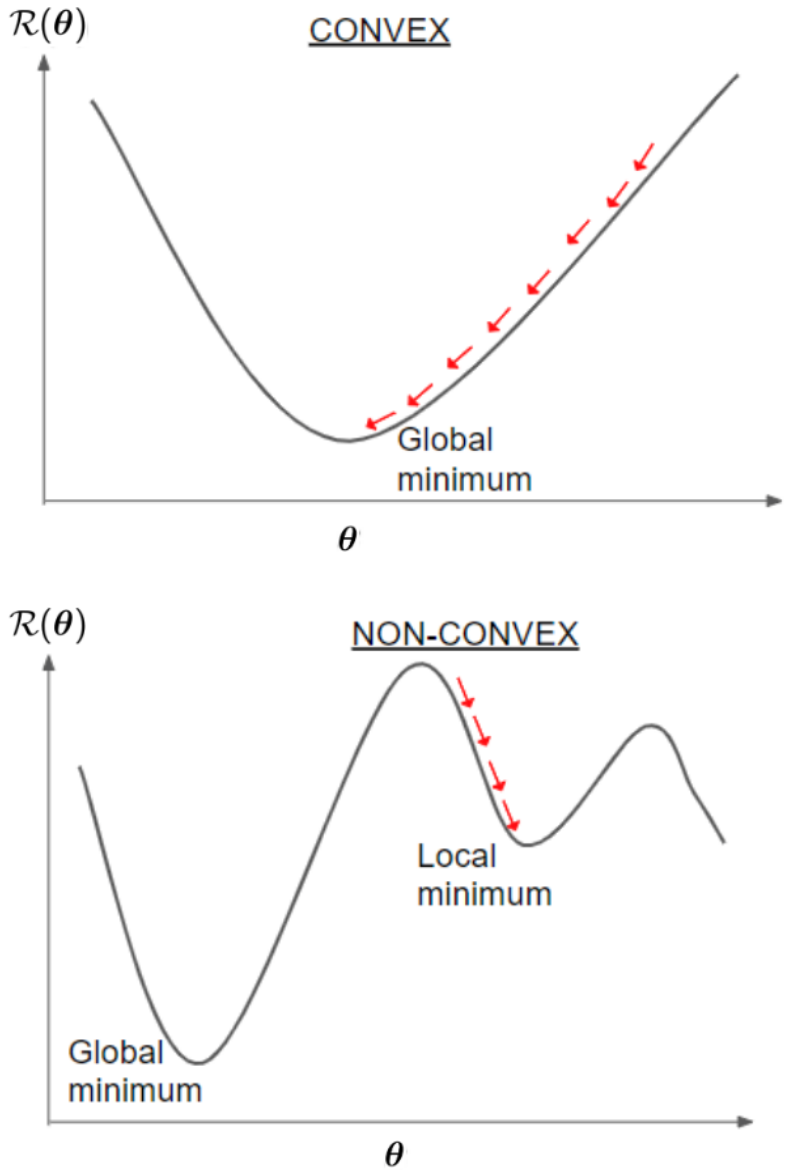
$$\boldsymbol{\theta}^{[t+1]} = \boldsymbol{\theta}^{[t]} - \alpha \mathbf{g},$$

- α is called **step size** or **learning rate**.



GRADIENT DESCENT AND OPTIMALITY

- GD is a greedy algorithm: In every iteration, it makes locally optimal moves.
- If $\mathcal{R}_{\text{emp}}(\boldsymbol{\theta})$ is **convex** and **differentiable**, and its gradient is Lipschitz continuous, GD is guaranteed to converge to the global minimum (for small enough step-size).
- However, if $\mathcal{R}_{\text{emp}}(\boldsymbol{\theta})$ has multiple local optima and/or saddle points, GD might only converge to a stationary point (other than the global optimum), depending on the starting point.



GRADIENT DESCENT AND OPTIMALITY / 2

Note: It might not be that bad if we do not find the global optimum:

- We don't optimize the (theoretical) risk, but only an approximate version, i.e. the empirical risk.
- For very flexible models, aggressive optimization might overfitting.
- Early-stopping might even increase generalization performance.

LEARNING RATE

The step-size α plays a key role in the convergence of the algorithm.

If the step size is too small, the training process may converge **very** slowly (see left image). If the step size is too large, the process may not converge, because it **jumps** around the optimal point (see right image).

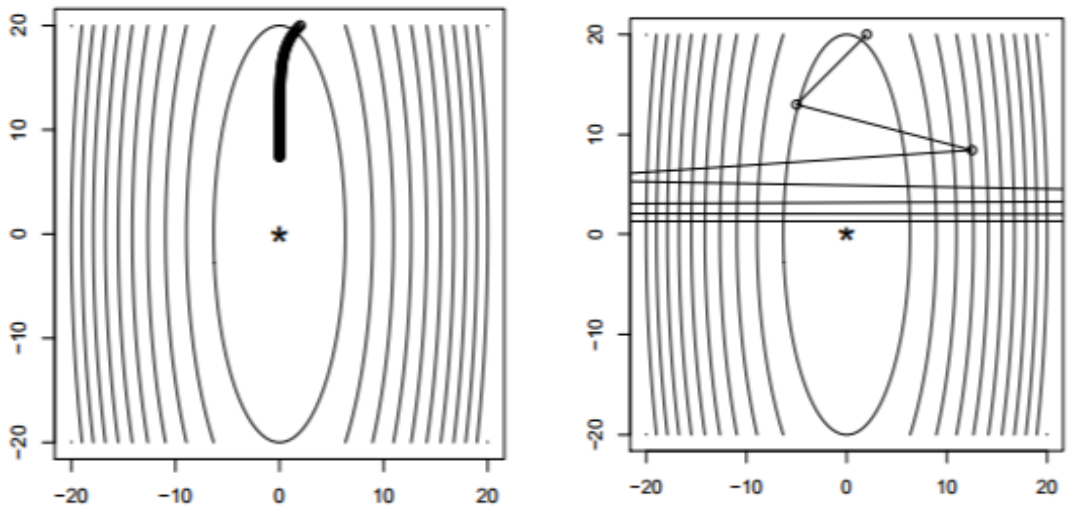


Figure: Small stepsize (left) vs. large stepsize (right) (Tibshirani, 2019)

LEARNING RATE

So far we have assumed a fixed value of α in every iteration:

$$\alpha^{[t]} = \alpha \quad \forall t = \{1, \ldots, T\}$$

However, it makes sense to adapt α in every iteration:

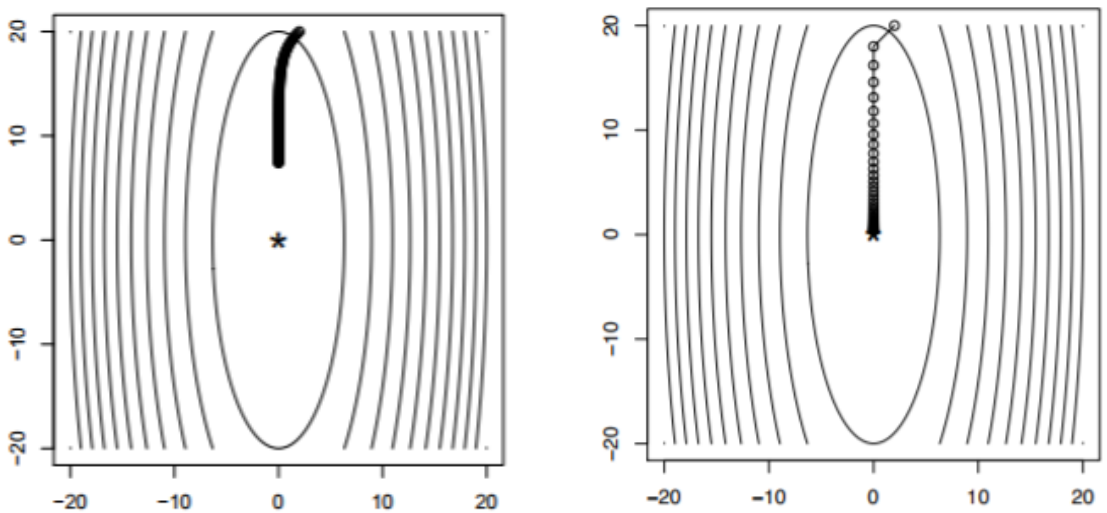


Figure: Steps of gradient descent for $\mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) = 10\,\theta_1^2 + 0.5\,\theta_2^2$. Left: 100 steps for with a fixed learning rate. Right: 40 steps with an adaptive learning rate (Tibshirani, 2019).

WEIGHT INITIALIZATION

- Weights (and biases) of an NN must be initialized in GD.
- We somehow must "break symmetry" – which would happen in full-0-initialization. If two neurons (with the same activation) are connected to the same inputs and have the same initial weights, then both neurons will have the same gradient update and learn the same features.
- Weights are typically drawn from a uniform a Gaussian distribution (both centered at 0 with a small variance).
- Two common initialization strategies are 'Glorot initialization' and 'He initialization' which tune the variance of these distributions based on the topology of the network.

STOCHASTIC GRADIENT DESCENT

GD for ERM was:

$$\boldsymbol{\theta}^{[t+1]} = \boldsymbol{\theta}^{[t]} - \alpha \cdot \frac{1}{n} \cdot \sum_{i=1}^{n} \nabla_\theta L \left(y^{(i)}, f(\mathbf{x}^{(i)} \mid \boldsymbol{\theta}^{[t]}) \right)$$

- Using the entire training set in GD to is called **batch** or **deterministic** or **offline training**. This can be computationally costly or impossible, if data does not fit into memory.
- **Idea:** Instead of letting the sum run over the whole dataset, use small stochastic subsets (**minibatches**), or only a single $\mathbf{x}^{(i)}$.
- If batches are uniformly sampled from $\mathcal{D}_{\text{train}}$, our stochastic gradient is in expectation the batch gradient $\nabla_\theta \mathcal{R}_{\text{emp}}(\boldsymbol{\theta})$.
- The gradient w.r.t. a single $\mathbf{x}$ is fast to compute but not reliable. But its often used in a theoretical analysis of SGD.

$\rightarrow$ We have a **stochastic**, noisy version of the batch GD.

STOCHASTIC GRADIENT DESCENT / 2

SGD on function $1.25(x_1 + 6)^2 + (x_2 - 8)^2$.

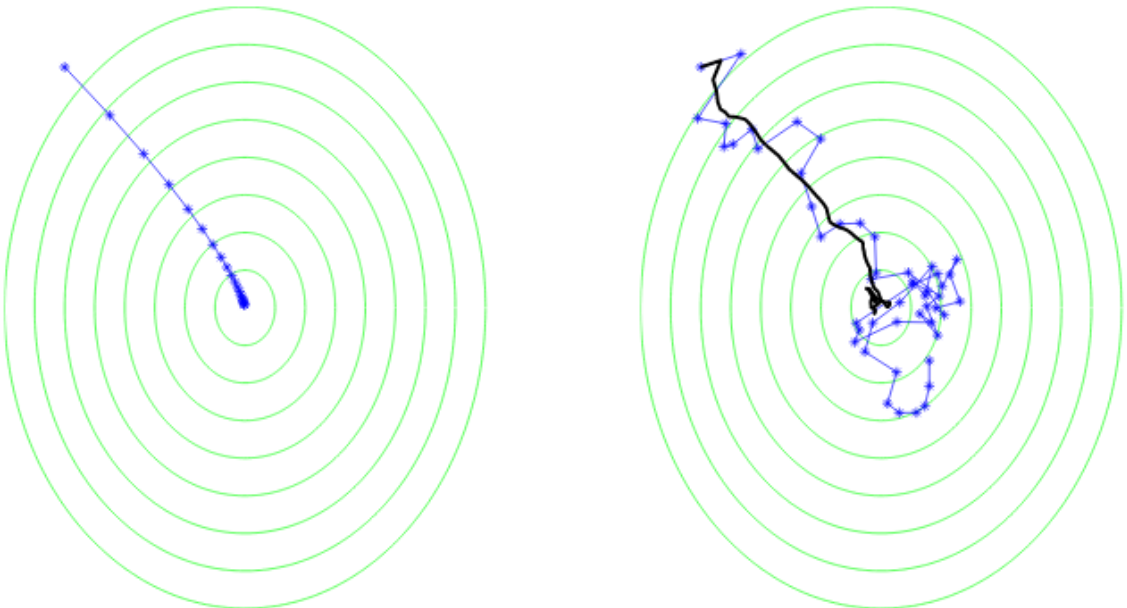


Figure: Left = GD, right = SGD. The black line is a smoothed θ
(Shalev-Shwartz & Ben-David, 2022)

STOCHASTIC GRADIENT DESCENT

Algorithm Basic SGD pseudo code

Initialize parameter vector $\boldsymbol{\theta}^{[0]}$
$t \leftarrow 0$
while stopping criterion not met **do**
    Randomly shuffle data and partition into minibatches $J_1, ..., J_K$ of size m
    for $k \in \{1, ..., K\}$ **do**
        $t \leftarrow t + 1$
        Compute gradient estimate with J_k: $\hat{g}^{[t]} \leftarrow \frac{1}{m} \sum_{i \in J_k} \nabla_\theta L(y^{(i)}, f(\mathbf{x}^{(i)} \mid \boldsymbol{\theta}^{[t-1]}))$
        Apply update: $\boldsymbol{\theta}^{[t]} \leftarrow \boldsymbol{\theta}^{[t-1]} - \alpha \hat{g}^{[t]}$
    end for
end while

- A full SGD pass over data is an **epoch**.
- Minibatch sizes are typically between 50 and 1000.

STOCHASTIC GRADIENT DESCENT / 2

- SGD is the most used optimizer in ML and especially in DL.
- We usually have to add a considerable amount of tricks to SGD to make it really efficient (e.g. momentum). More on this later.
- SGD with (small) batches has a high variance, although is unbiased. Hence, the LR α is smaller than in the batch mode.
- When LR is slowly decreased, SGD converges to local minimum.
- Recent results indicate that SGD often leads to better generalization than GD, and may result in indirect regularization.

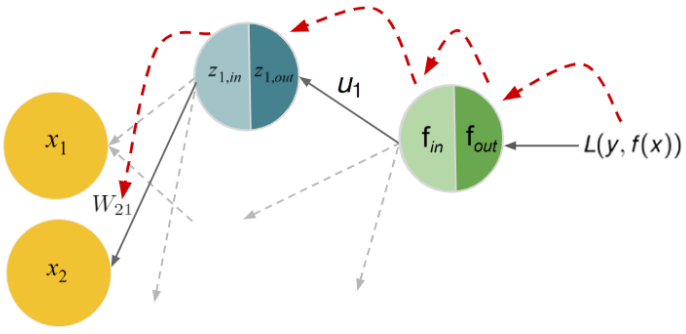
REFERENCES

Tibshirani, R. (2019, September 11). *Convex Optimization, Lecture 5: September 11*. Carnegie Mellon University Statistics & Data Science. `https://www.stat.cmu.edu/~ryantibs/convexopt/scribes/grad-descent-scribed.pdf`

Shalev-Shwartz, S., & Ben-David, S. (2022). *Understanding machine learning: From theory to algorithms*. Cambridge University Press.

Deep Learning

Chain Rule and Computational Graphs



Learning goals

- Chain rule of calculus
- Computational graphs

CHAIN RULE OF CALCULUS

- The chain rule can be used to compute derivatives of the composition of two or more functions.
- Let $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{y} \in \mathbb{R}^n$,
  $g : \mathbb{R}^m \to \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}$.
- If $\mathbf{y} = g(\mathbf{x})$ and $z = f(\mathbf{y})$, the chain rule yields:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \cdot \frac{\partial y_j}{\partial x_i}$$

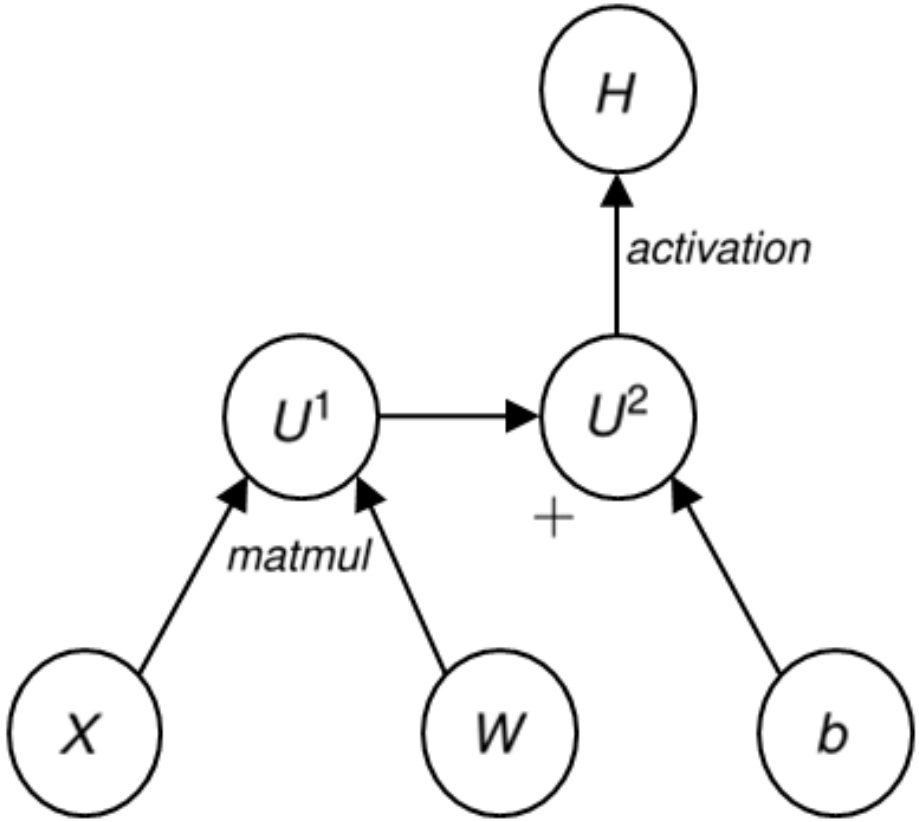
  or, in vector notation:

$$\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^{\top} \nabla_{\mathbf{y}} z,$$

  where $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ is the $(n \times m)$ Jacobian matrix of g.

COMPUTATIONAL GRAPHS

- CGs are nested expresssions, visualized as graphs.
- Each node is a variable, either an input or derived.
- Derived variables are functions applied to other variables.



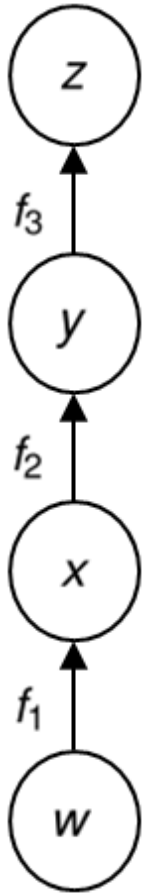
source : Goodfellow et al. (2016)

Figure: The computational graph for the expression $H = \sigma(XW + B)$ with activation function $\sigma(\cdot)$.

CHAIN RULE OF CALCULUS: EXAMPLE 1

- Suppose we have the following computational graph.
- To compute the derivative of $\frac{\partial z}{\partial w}$ we need to recursively apply the chain rule. That is:

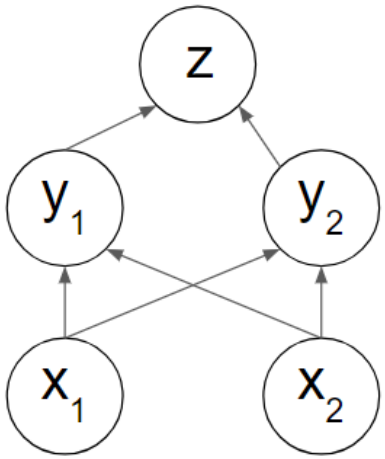
$$\begin{aligned}\frac{\partial z}{\partial w} &= \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x} \cdot \frac{\partial x}{\partial w} \\ &= f_3'(y) \cdot f_2'(x) \cdot f_1'(w) \\ &= f_3'(f_2(f_1(w))) \cdot f_2'(f_1(w)) \cdot f_1'(w)\end{aligned}$$



source : Goodfellow et al. (2016)

Figure: A computational graph, such that $x = f_1(w)$, $y = f_2(x)$ and $z = f_3(y)$.

CHAIN RULE OF CALCULUS: EXAMPLE 2



To compute $\nabla_{\mathbf{x}} z$, we apply the chain rule

- $\frac{\partial z}{\partial x_1} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_1} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x_1} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x_1}$
- $\frac{\partial z}{\partial x_2} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_2} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x_2} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x_2}$

Therefore, the gradient of z w.r.t $\mathbf{x}$ is

- $\nabla_{\mathbf{x}} z = \begin{bmatrix} \frac{\partial z}{\partial x_1} \\ \frac{\partial z}{\partial x_2} \end{bmatrix} = \underbrace{\begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} \end{bmatrix}}_{(\frac{\partial \mathbf{y}}{\partial \mathbf{x}})^\top} \underbrace{\begin{bmatrix} \frac{\partial z}{\partial y_1} \\ \frac{\partial z}{\partial y_2} \end{bmatrix}}_{\nabla_{\mathbf{y}} z} = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^\top \nabla_{\mathbf{y}} z$

COMPUTATIONAL GRAPH: NEURAL NET

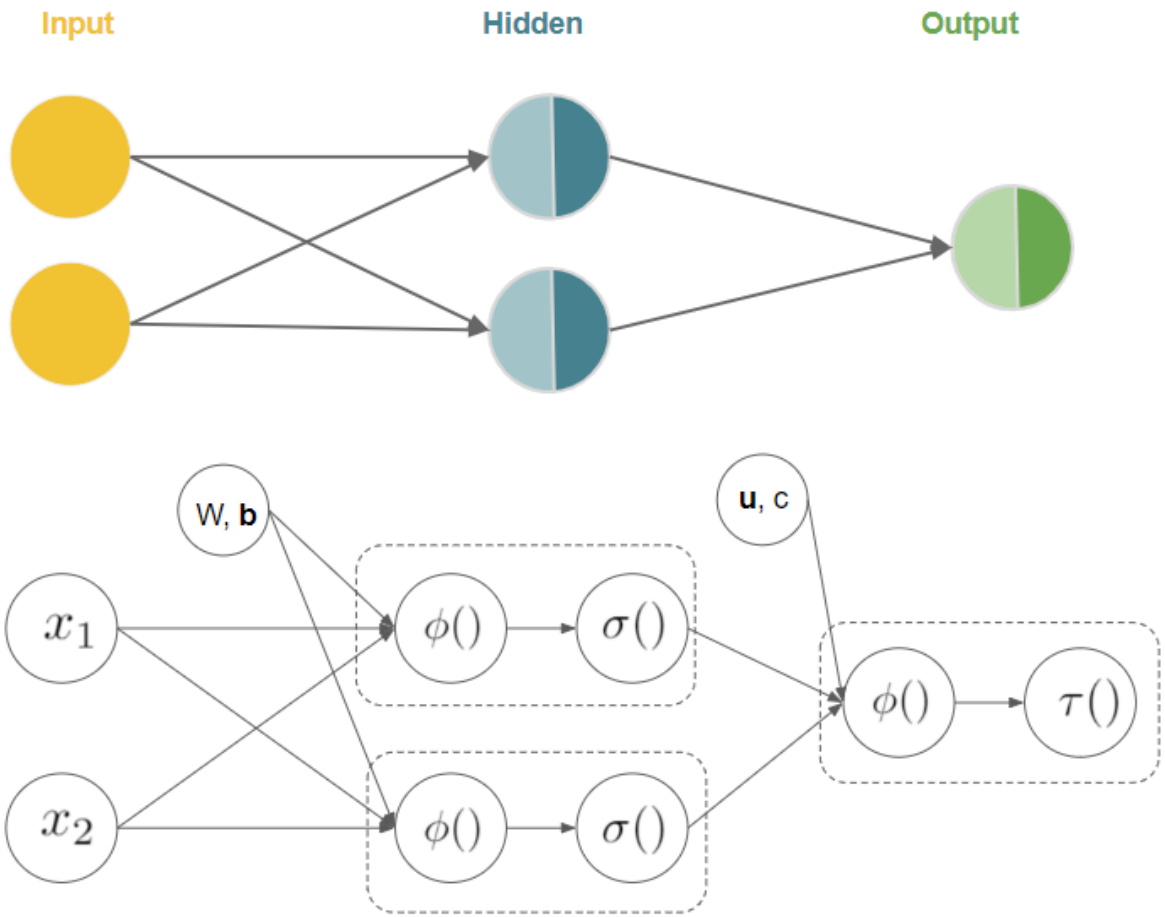
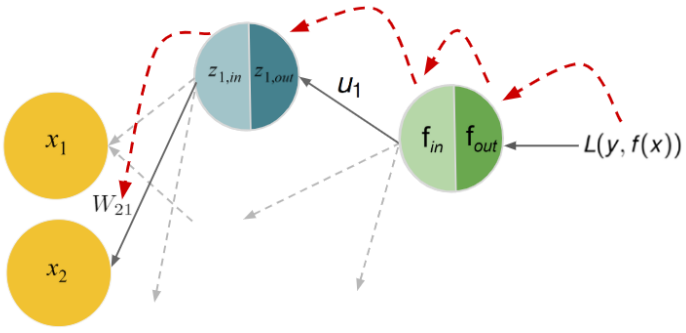


Figure: A neural network can be seen as a computational graph. ϕ is the weighted sum and σ and τ are the activations.
Note: In contrast to the top figure, the arrows in the computational graph below merely indicate **dependence**, not weights.

Deep Learning

Basic Backpropagation 1



Learning goals

- Forward and backward passes
- Chain rule
- Details of backprop

BACKPROPAGATION: BASIC IDEA

We would like to run ERM by GD on:

$$\mathcal{R}_{\text{emp}}(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} L\left(y^{(i)}, f\left(\mathbf{x}^{(i)} \mid \boldsymbol{\theta}\right)\right)$$

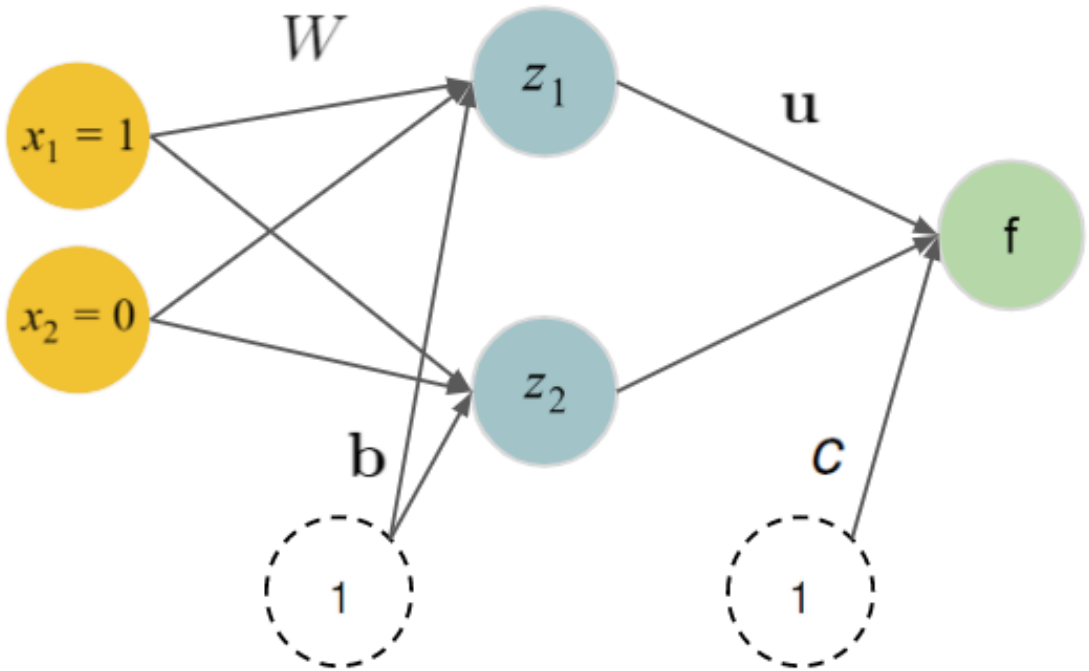
Backprop training of NNs runs in 2 alternating steps, for one $\mathbf{x}$:

1. **Forward pass:** Inputs flow through model to outputs. We then compute the observation loss. We covered that.
2. **Backward pass:** Loss flows backwards to update weights so error is reduced, as in GD.

We will see: This is simply (S)GD in disguise, cleverly using the chain rule, so we can reuse a lot of intermediate results.

XOR EXAMPLE

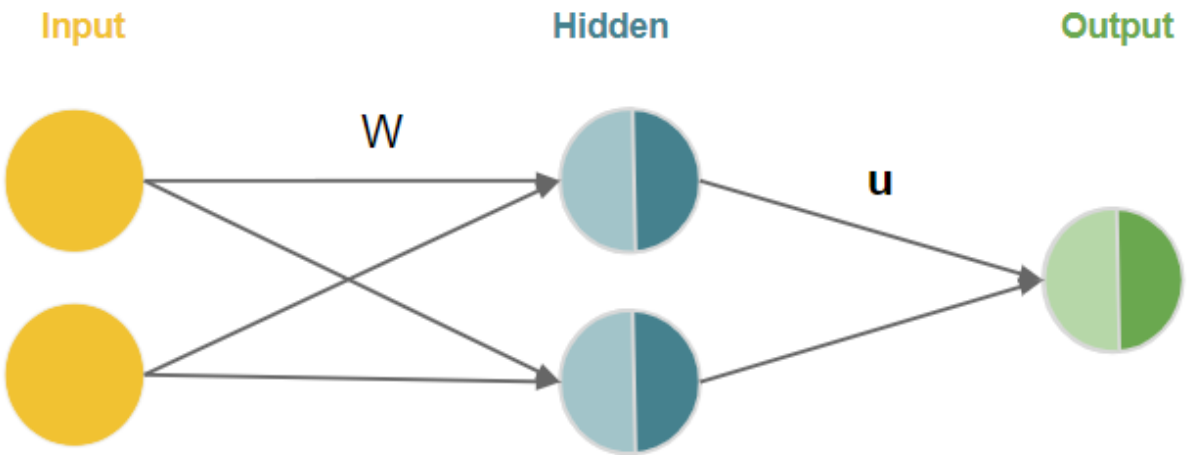
- As activations (hidden and outputs) we use the logistic.
- We run one FP and BP on $\mathbf{x} = (1, 0)^T$ with $y = 1$.
- We use L2 loss between 0-1 labels and the predicted probabilities. This is a bit uncommon, but computations become simpler.



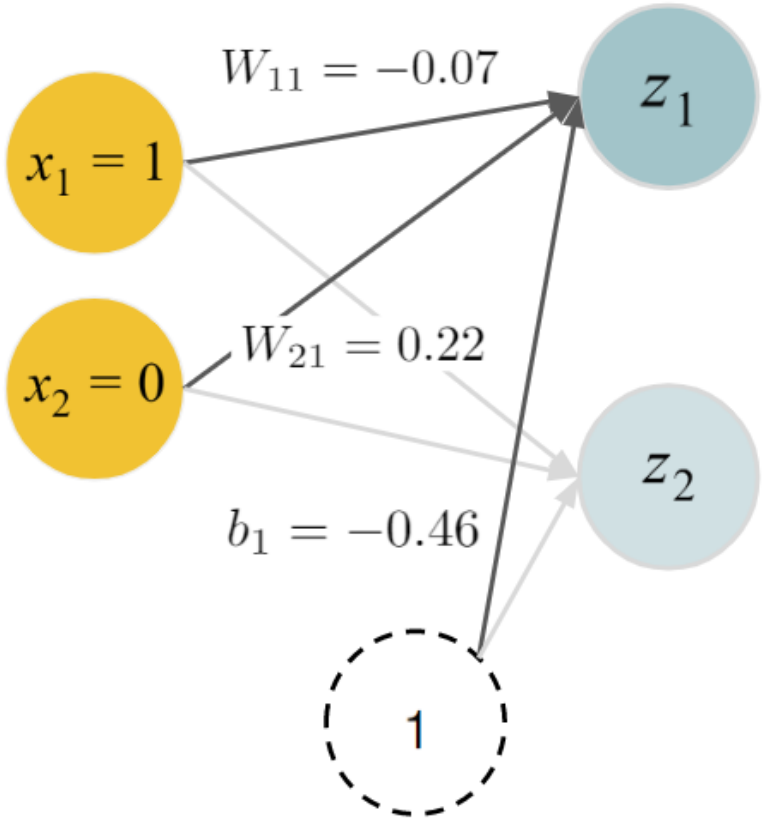
Note: We will only show rounded decimals.

FORWARD PASS

- We will divide the FP into four steps:
  - the inputs of z_j: $\boldsymbol{z_{j,in}}$
  - the activations of z_j: $\boldsymbol{z_{j,out}}$
  - the input of f: $\boldsymbol{f_{in}}$
  - and finally the activation of f: $\boldsymbol{f_{out}}$



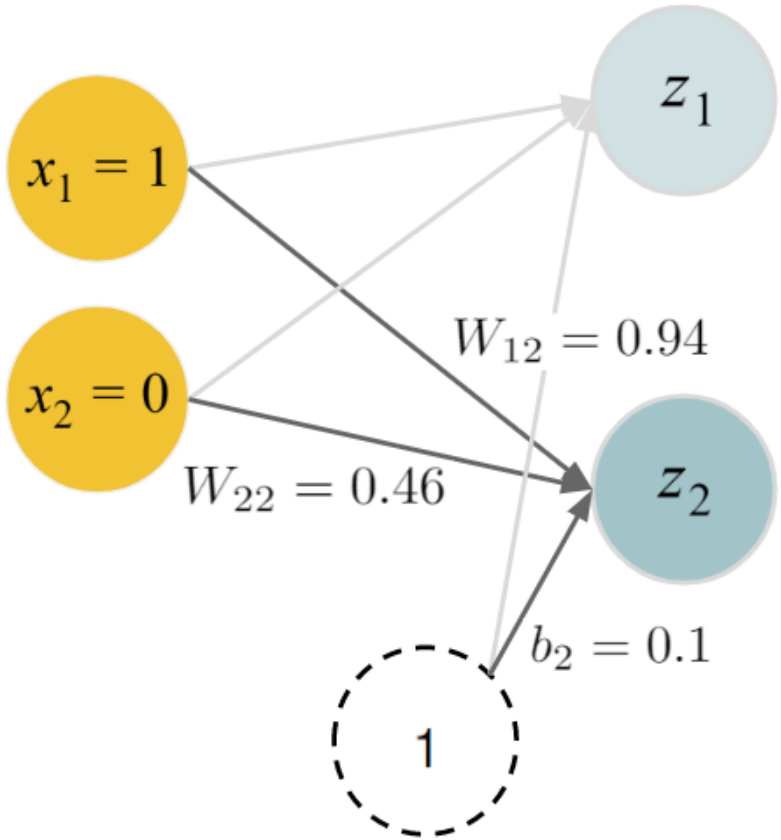
FORWARD PASS / 2



$$z_{1,in} = \mathbf{W}_1^T \mathbf{x} + b_1 = 1 \cdot (-0.07) + 0 \cdot 0.22 + 1 \cdot (-0.46) = -0.53$$

$$z_{1,out} = \sigma(z_{1,in}) = \frac{1}{1 + \exp(-(-0.53))} = 0.3705$$

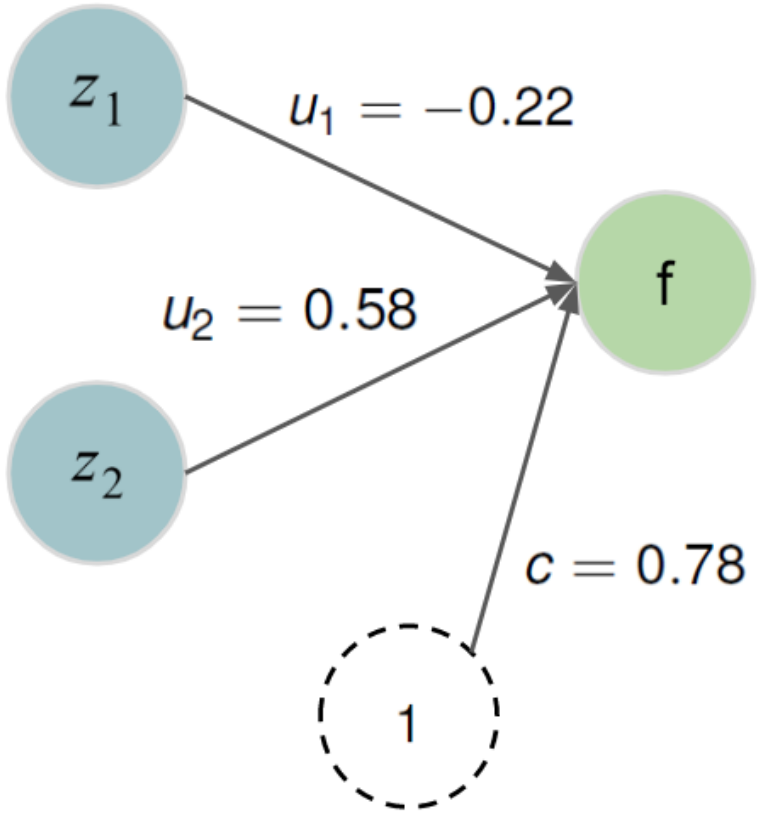
FORWARD PASS / 3



$$z_{2,in} = \mathbf{W}_2^T \mathbf{x} + b_2 = 1 \cdot 0.94 + 0 \cdot 0.46 + 1 \cdot 0.1 = 1.04$$

$$z_{2,out} = \sigma\left(z_{2,in}\right) = \frac{1}{1 + \exp(-1.04)} = 0.7389$$

FORWARD PASS / 4



$$f_{in} = \boldsymbol{u}^T \boldsymbol{z} + c = 0.3705 \cdot (-0.22) + 0.7389 \cdot 0.58 + 1 \cdot 0.78 = 1.1122$$

$$f_{out} = \tau(f_{in}) = \frac{1}{1 + \exp(-1.1122)} = 0.7525$$

FORWARD PASS / 5

- The FP predicted $f_{out} = 0.7525$
- Now, we compare the prediction $f_{out} = 0.7525$ and the true label $y = 1$ using the L2-loss:

$$\begin{aligned} L(y, f(\mathbf{x})) &= \frac{1}{2}(y - f(\mathbf{x}^{(i)} \mid \boldsymbol{\theta}))^2 = \frac{1}{2}(y - f_{out})^2 \\ &= \frac{1}{2}(1 - 0.7525)^2 = 0.0306 \end{aligned}$$

- The calculation of the gradient is performed backwards (starting from the output layer), so that results can be reused.

Check the arithmetic: a slip in the forward pass

Redo LMU's output step with a calculator:

$$f_{\text{in}} = 0.3705\cdot(-0.22)+0.7389\cdot 0.58+0.78 = -0.0815+0.4286+0.78 = \mathbf{1.1271}$$

	LMU's slides	**correct**
f_{in}	1.1122	1.1271
$f_{\text{out}} = \sigma(f_{\text{in}})$	0.7525	0.7553
$L = \frac{1}{2}(1 - f_{\text{out}})^2$	0.0306	0.0299
loss after one update	0.0284	0.0282

The method is right; only the numbers inherit the slip. The next pages keep LMU's 0.7525, so every step still checks out against the one before it. The lesson is unchanged: one backward pass and one update lower the loss.

BACKWARD PASS

The main ingredients of the backward pass are:

- to reuse the results of the forward pass
  (here: $z_{j,in}, z_{j,out}, f_{in}, f_{out}$)
- reuse the intermediate results from the chain rule
- the derivative of the activations and some affine functions

BACKWARD PASS / 2

- Let's start to update u_1. We recursively apply the chain rule:

$$\frac{\partial L(y, f(\mathbf{x}))}{\partial u_1} = \frac{\partial L(y, f(\mathbf{x}))}{\partial f_{out}} \cdot \frac{\partial f_{out}}{\partial f_{in}} \cdot \frac{\partial f_{in}}{\partial u_1}$$

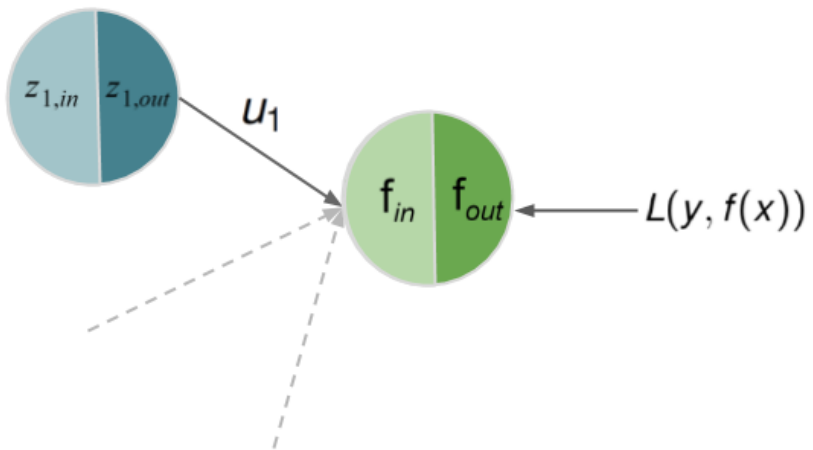


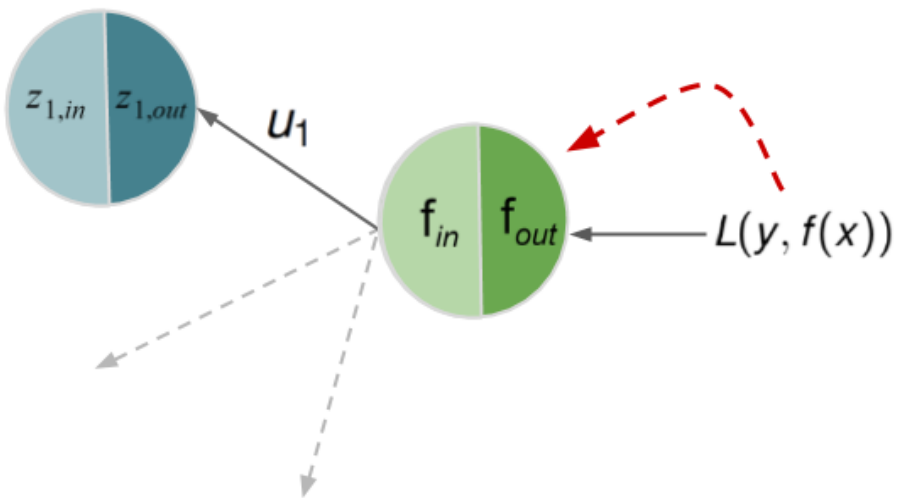
Figure: Snippet from our NN, with backward path for u_1.

BACKWARD PASS / 3

- 1st step: The derivative of L2 is easy; we know f_{out} from FP.

$$\frac{\partial L(y, f(\mathbf{x}))}{\partial f_{out}} = \frac{d}{\partial f_{out}} \frac{1}{2}(y - f_{out})^2 = -\underbrace{(y - f_{out})}_{\hat{=}\text{residual}}$$

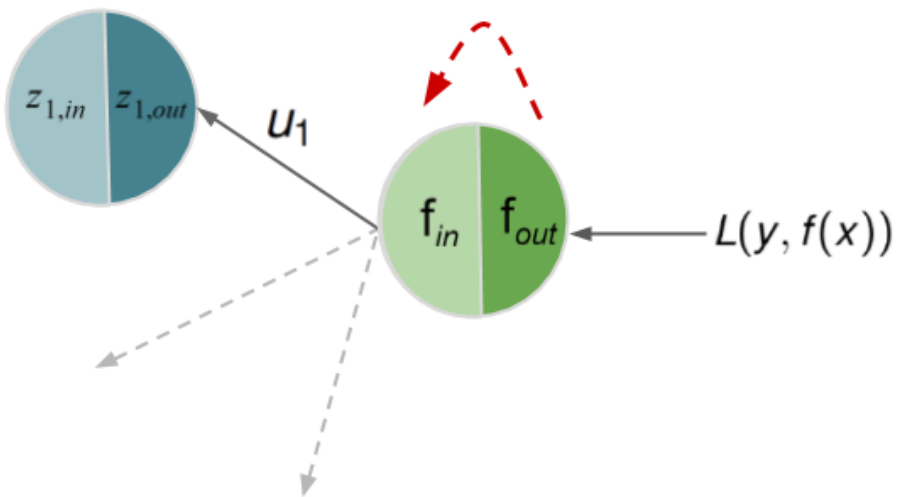
$$= -(1 - 0.7525) = -0.2475$$



BACKWARD PASS / 4

- 2nd step. $f_{out} = \sigma(f_{in})$, use rule for σ', use f_{in} from FP.

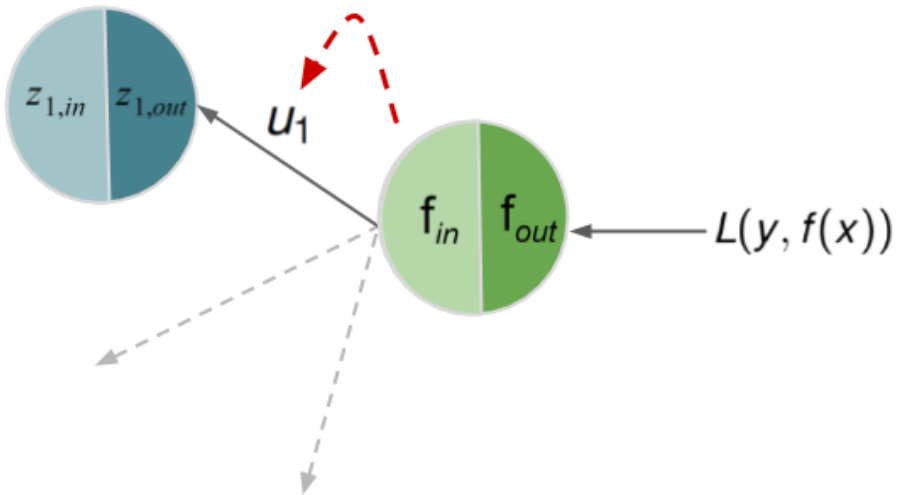
$$\begin{aligned} \frac{\partial f_{out}}{\partial f_{in}} &= \sigma(f_{in}) \cdot (1 - \sigma(f_{in})) \\ &= 0.7525 \cdot (1 - 0.7525) = 0.1862 \end{aligned}$$



BACKWARD PASS / 5

- 3rd step. Derivative of the linear input is easy; use $z_{1,out}$ from FP.

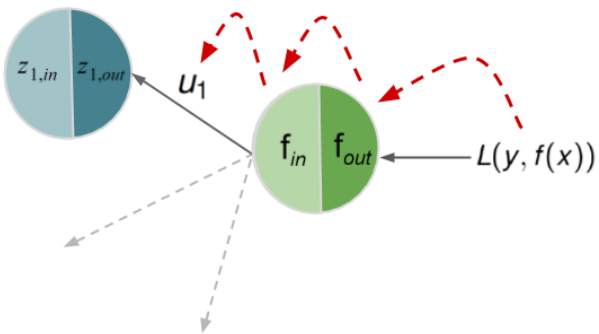
$$\frac{\partial f_{in}}{\partial u_1} = \frac{\partial(u_1 \cdot z_{1,out} + u_2 \cdot z_{2,out} + c \cdot 1)}{\partial u_1} = z_{1,out} = 0.3705$$



BACKWARD PASS / 6

- Plug it together:

$$\begin{aligned}\frac{\partial L(y, f(\mathbf{x}))}{\partial u_1} &= \frac{\partial L(y, f(\mathbf{x}))}{\partial f_{out}} \cdot \frac{\partial f_{out}}{\partial f_{in}} \cdot \frac{\partial f_{in}}{\partial u_1} \\ &= -0.2475 \cdot 0.1862 \cdot 0.3705 = -0.0171\end{aligned}$$



- With LR $\alpha = 0.5$:

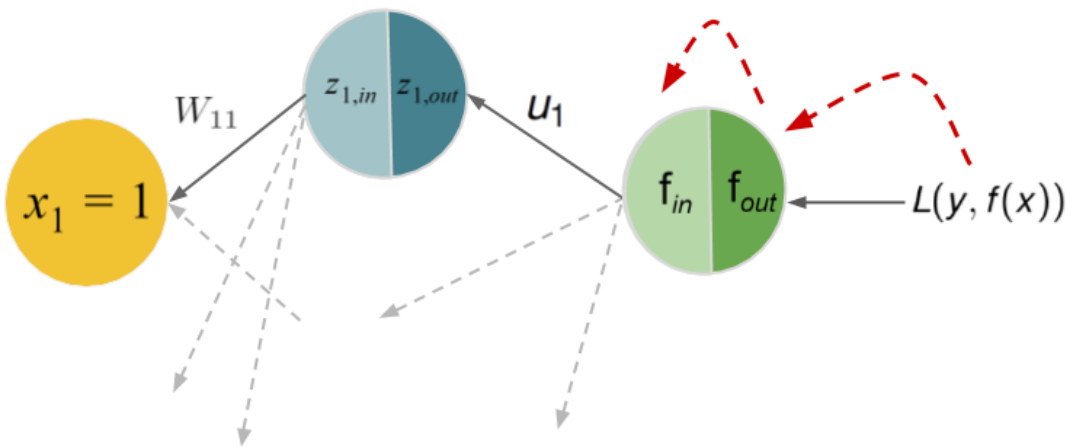
$$\begin{aligned}u_1^{[t+1]} &= u_1^{[t]} - \alpha \cdot \frac{\partial L(y, f(\mathbf{x}))}{\partial u_1} \\ &= -0.22 - 0.5 \cdot (-0.0171) = -0.2115\end{aligned}$$

BACKWARD PASS / 7

- Now for W_{11}:

$$\frac{\partial L(y, f(\mathbf{x}))}{\partial W_{11}} = \frac{\partial L(y, f(\mathbf{x}))}{\partial f_{out}} \cdot \frac{\partial f_{out}}{\partial f_{in}} \cdot \frac{\partial f_{in}}{\partial z_{1,out}} \cdot \frac{\partial z_{1,out}}{\partial z_{1,in}} \cdot \frac{\partial z_{1,in}}{\partial W_{11}}$$

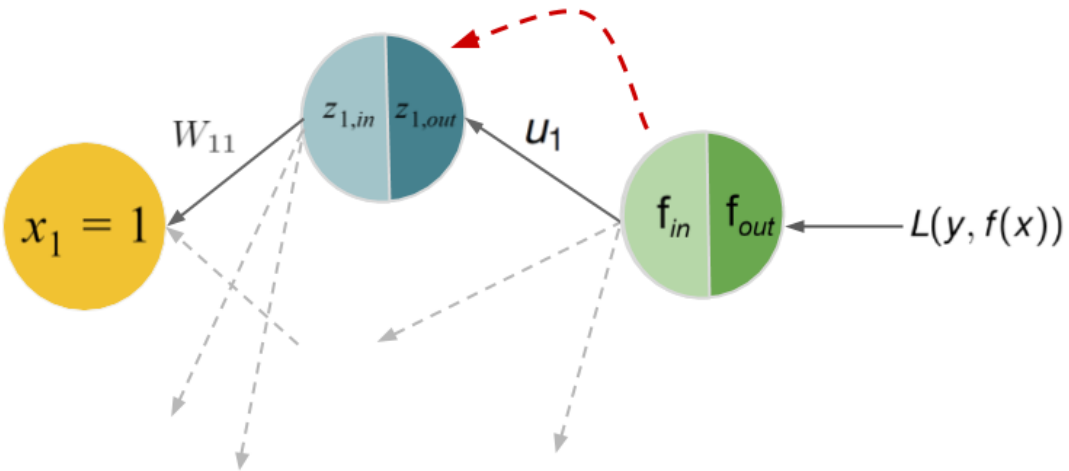
- We know $\frac{\partial L(y,f(\mathbf{x}))}{\partial f_{out}}$ and $\frac{\partial f_{out}}{\partial f_{in}}$ from BP for u_1.



BACKWARD PASS / 8

- $f_{in} = u_1 \cdot z_{1,out} + u_2 \cdot z_{2,out} + c \cdot 1$ is linear, easy and we know u_1 :

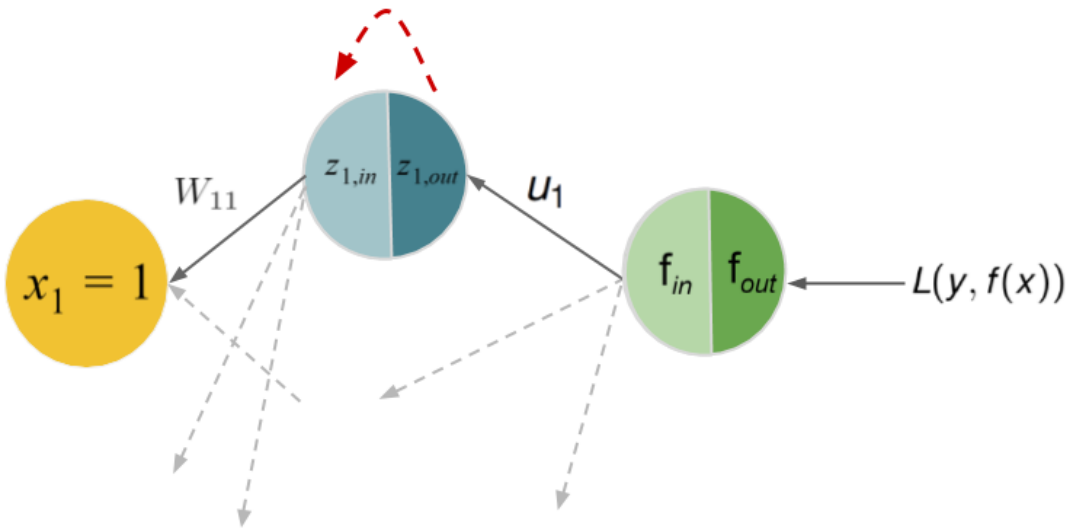
$$\frac{\partial f_{in}}{\partial z_{1,out}} = u_1 = -0.22$$



BACKWARD PASS / 9

- Next. Use rule for σ' and FP results:

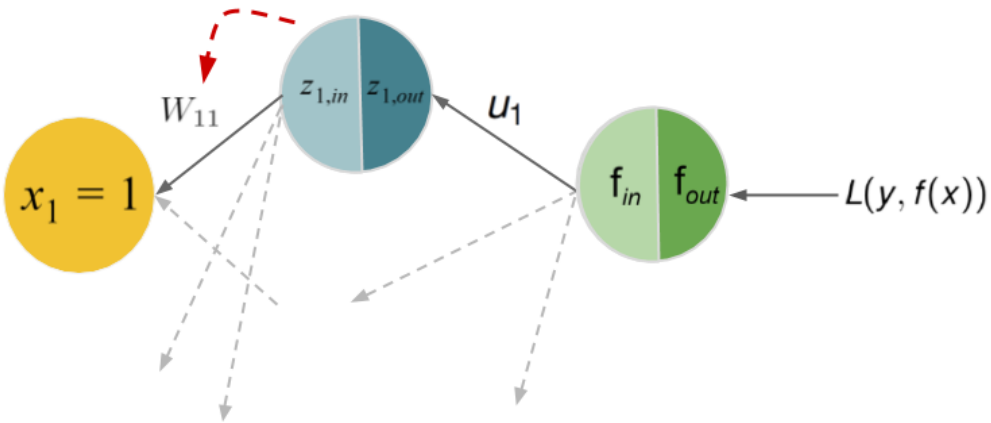
$$\begin{aligned} \frac{\partial z_{1,out}}{\partial z_{1,in}} &= \sigma(z_{1,in}) \cdot (1 - \sigma(z_{1,in})) \\ &= 0.3705 \cdot (1 - 0.3705) = 0.2332 \end{aligned}$$



BACKWARD PASS / 10

- $z_{1,in} = x_1 \cdot W_{11} + x_2 \cdot W_{21} + b_1 \cdot 1$ is linear and depends on inputs:

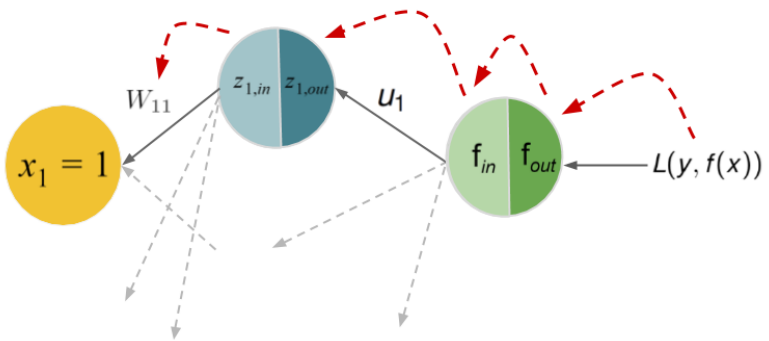
$$\frac{\partial z_{1,in}}{\partial W_{11}} = x_1 = 1$$



BACKWARD PASS / 11

- Plugging together:

$$\begin{aligned}\frac{\partial L(y, f(\mathbf{x}))}{\partial W_{11}} &= \frac{\partial L(y, f(\mathbf{x}))}{\partial f_{out}} \cdot \frac{\partial f_{out}}{\partial f_{in}} \cdot \frac{\partial f_{in}}{\partial z_{1,out}} \cdot \frac{\partial z_{1,out}}{\partial z_{1,in}} \cdot \frac{\partial z_{1,in}}{\partial W_{11}} \\ &= (-0.2475) \cdot 0.1862 \cdot (-0.22) \cdot 0.2332 \cdot 1 \\ &= 0.0024\end{aligned}$$



- Full SGD update:

$$\begin{aligned}W_{11}^{[t+1]} &= W_{11}^{[t]} - \alpha \cdot \frac{\partial L(y, f(\mathbf{x}))}{\partial W_{11}} \\ &= -0.07 - 0.5 \cdot 0.0024 = -0.0712\end{aligned}$$

RESULT

- We can do this for all weights:

$$\mathbf{W} = \begin{pmatrix} -0.0712 & 0.9426 \\ 0.22 & 0.46 \end{pmatrix}, \mathbf{b} = \begin{pmatrix} -0.4612 \\ 0.1026 \end{pmatrix},$$

$$\mathbf{u} = \begin{pmatrix} -0.2115 \\ 0.5970 \end{pmatrix} \text{ and } c = 0.8030.$$

- Yields $f(\mathbf{x} \mid \boldsymbol{\theta}^{[t+1]}) = 0.7615$ and loss $\frac{1}{2}(1 - 0.7615)^2 = 0.0284$.
- Before, we had $f(\mathbf{x} \mid \boldsymbol{\theta}^{[t]}) = 0.7525$ and higher loss 0.0306.

Now rinse and repeat. This was one training iter, we do thousands.

A cleaner residual: the output gradient is $f - y$

LMU used the **L2 loss** on class labels to keep the algebra simple. In practice, classification uses **cross-entropy** – and then the output-layer gradient collapses to something you already know:

$$\text{sigmoid} + \text{cross-entropy} \implies \frac{\partial L}{\partial f_{\text{in}}} = f - y$$

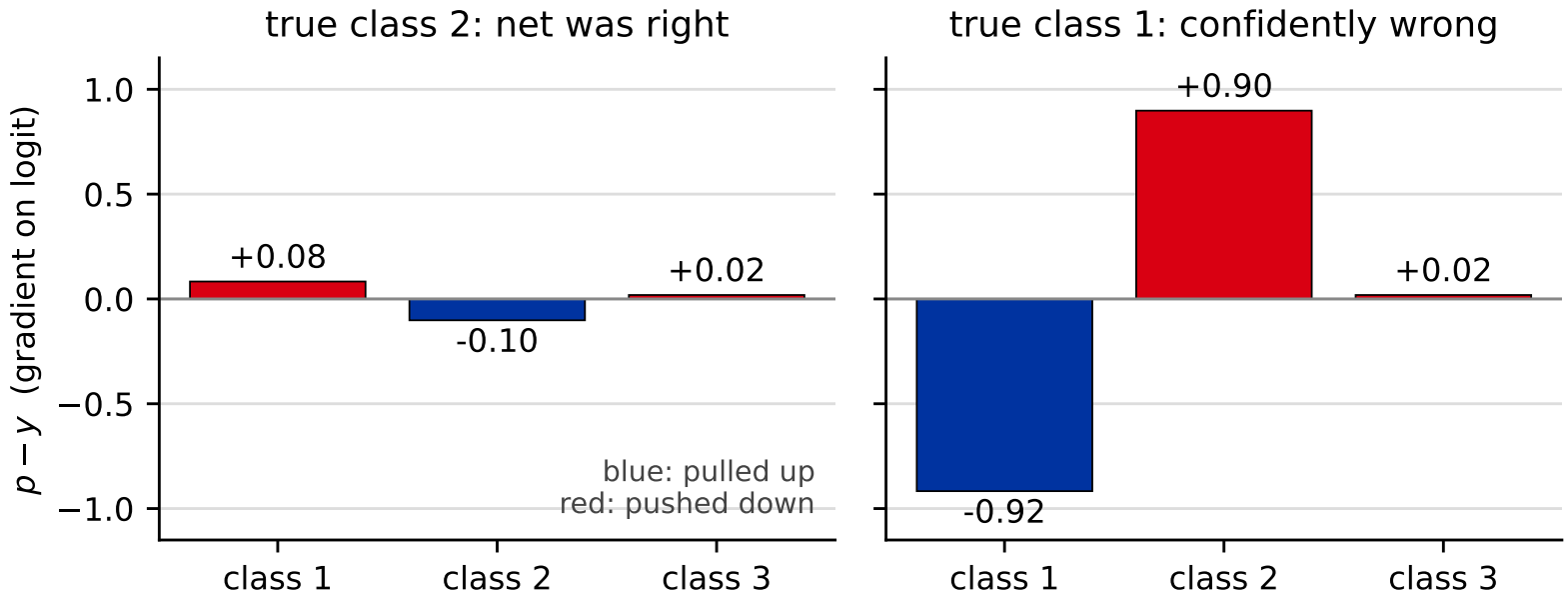
$$\text{softmax} + \text{cross-entropy} \implies \frac{\partial L}{\partial \mathbf{f}_{\text{in}}} = \mathbf{p} - \mathbf{y} \quad (\text{one-hot } \mathbf{y})$$

Why not L2? Its gradient keeps a $\sigma'(f_{\text{in}})$ factor, which is tiny exactly when the net is confidently wrong. Take $f = 0.01$ with $y = 1$: L2 gives $-(1 - 0.01) \cdot 0.01 \cdot 0.99 \approx -0.0098$, cross-entropy gives $f - y = -0.99$, a push a hundred times stronger.

The same "residual" from earlier in the course. Linear regression, logistic regression, and now a neural net: the output signal is **prediction** − **target**. Everything to its left is that residual, passed backward and scaled by local derivatives.

The softmax gradient as push and pull

The softmax example from the multilayer-nets deck: logits $(0.57,\ 2.95,\ -0.92)$ give $\mathbf{p} = (0.08,\ 0.90,\ 0.02)$. The gradient on the logits is $\mathbf{p} - \mathbf{y}$, and a descent step moves each logit *against* it:

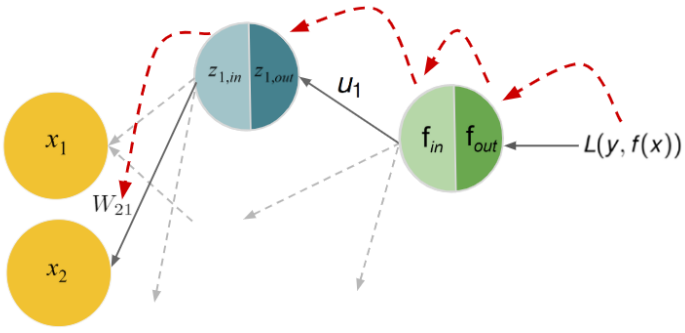


- The correct class is **pulled up** by $1 - p_{\text{correct}}$; every wrong class is **pushed down** by its own p. The pushes and the pull cancel: each row sums to 0.
- The force is exactly how wrong the net was: right and confident, a nudge; confidently wrong, a shove. A perfect prediction gets no gradient at all.

Intuition: Karpathy, *Becoming a Backprop Ninja* (2022).

Deep Learning

Basic Backpropagation 2



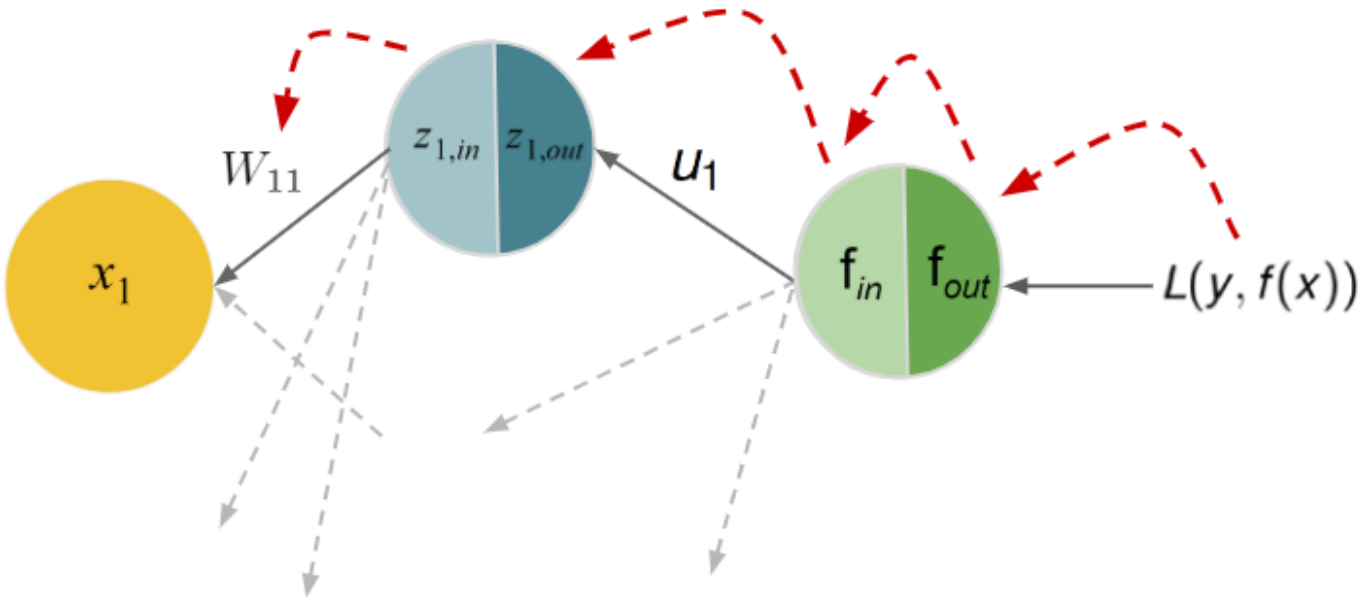
Learning goals

- Backprop formalism and recursion

BACKWARD COMPUTATION AND CACHING

In the XOR example, we computed:

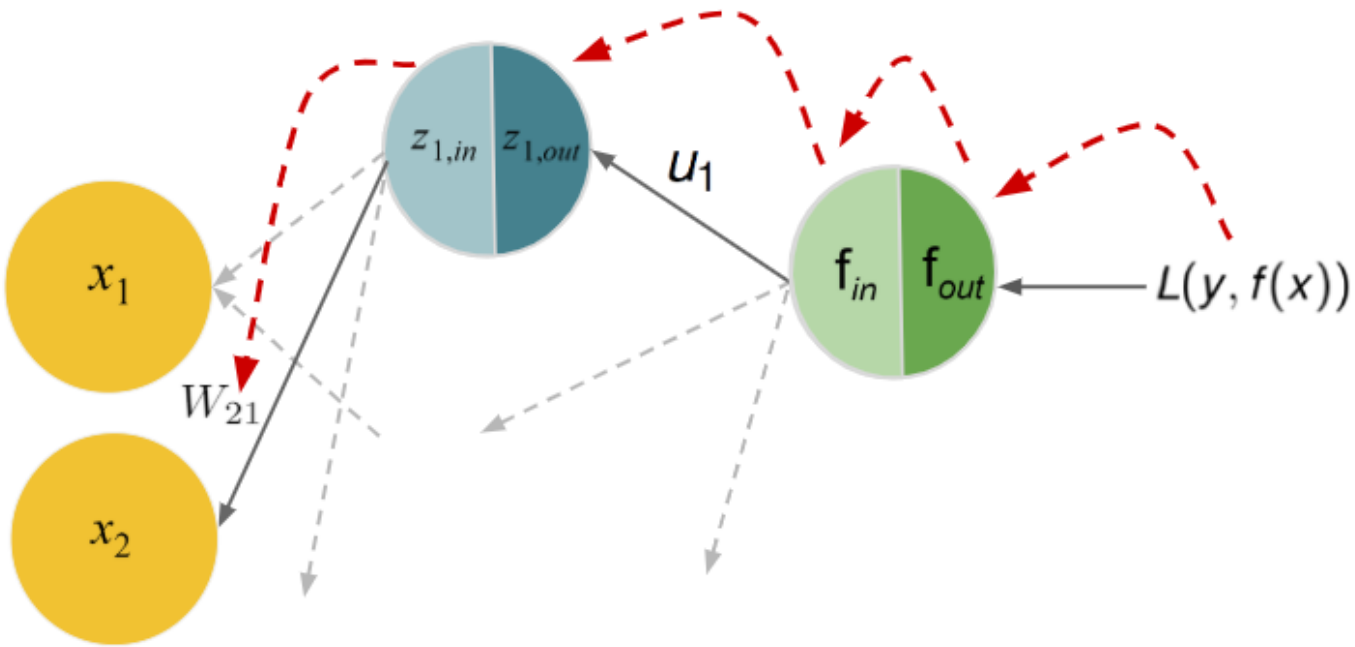
$$\frac{\partial L(y, f(\mathbf{x}))}{\partial W_{11}} = \frac{\partial L(y, f(\mathbf{x}))}{\partial f_{out}} \cdot \frac{\partial f_{out}}{\partial f_{in}} \cdot \frac{\partial f_{in}}{\partial z_{1,out}} \cdot \frac{\partial z_{1,out}}{\partial z_{1,in}} \cdot \frac{\partial z_{1,in}}{\partial W_{11}}$$



BACKWARD COMPUTATION AND CACHING

Next, let us compute:

$$\frac{\partial L(y, f(\mathbf{x}))}{\partial W_{21}} = \frac{\partial L(y, f(\mathbf{x}))}{\partial f_{out}} \cdot \frac{\partial f_{out}}{\partial f_{in}} \cdot \frac{\partial f_{in}}{\partial z_{1,out}} \cdot \frac{\partial z_{1,out}}{\partial z_{1,in}} \cdot \frac{\partial z_{1,in}}{\partial W_{21}}$$



BACKWARD COMPUTATION AND CACHING

- Examining the two expressions:

$$\frac{\partial L(y, f(\mathbf{x}))}{\partial W_{11}} = \frac{\partial L(y, f(\mathbf{x}))}{\partial f_{out}} \cdot \frac{\partial f_{out}}{\partial f_{in}} \cdot \frac{\partial f_{in}}{\partial z_{1,out}} \cdot \frac{\partial z_{1,out}}{\partial z_{1,in}} \cdot \frac{\partial z_{1,in}}{\partial W_{11}}$$

$$\frac{\partial L(y, f(\mathbf{x}))}{\partial W_{21}} = \frac{\partial L(y, f(\mathbf{x}))}{\partial f_{out}} \cdot \frac{\partial f_{out}}{\partial f_{in}} \cdot \frac{\partial f_{in}}{\partial z_{1,out}} \cdot \frac{\partial z_{1,out}}{\partial z_{1,in}} \cdot \frac{\partial z_{1,in}}{\partial W_{21}}$$

- Significant overlap / redundancy in the two expressions.
- **Again**: Let's call this subexpression δ_1 and cache it.

$$\delta_1 = \frac{\partial L(y, f(\mathbf{x}))}{\partial z_{1,in}} = \frac{\partial L(y, f(\mathbf{x}))}{\partial f_{out}} \cdot \frac{\partial f_{out}}{\partial f_{in}} \cdot \frac{\partial f_{in}}{\partial z_{1,out}} \cdot \frac{\partial z_{1,out}}{\partial z_{1,in}}$$

$$\frac{\partial L(y, f(\mathbf{x}))}{\partial W_{11}} = \delta_1 \cdot \frac{\partial z_{1,in}}{\partial W_{11}} \quad \text{and} \quad \frac{\partial L(y, f(\mathbf{x}))}{\partial W_{21}} = \delta_1 \cdot \frac{\partial z_{1,in}}{\partial W_{21}}$$

- δ_1 can also be seen as an **error signal** that represents how much the loss L changes when the input $z_{1,in}$ changes.

BACKPROP: RECURSION

- Let us now derive a general formulation of backprop.
- The neurons in layers $i-1$, i and $i+1$ are indexed by j, k and m, respectively.
- The output layer will be referred to as layer O.

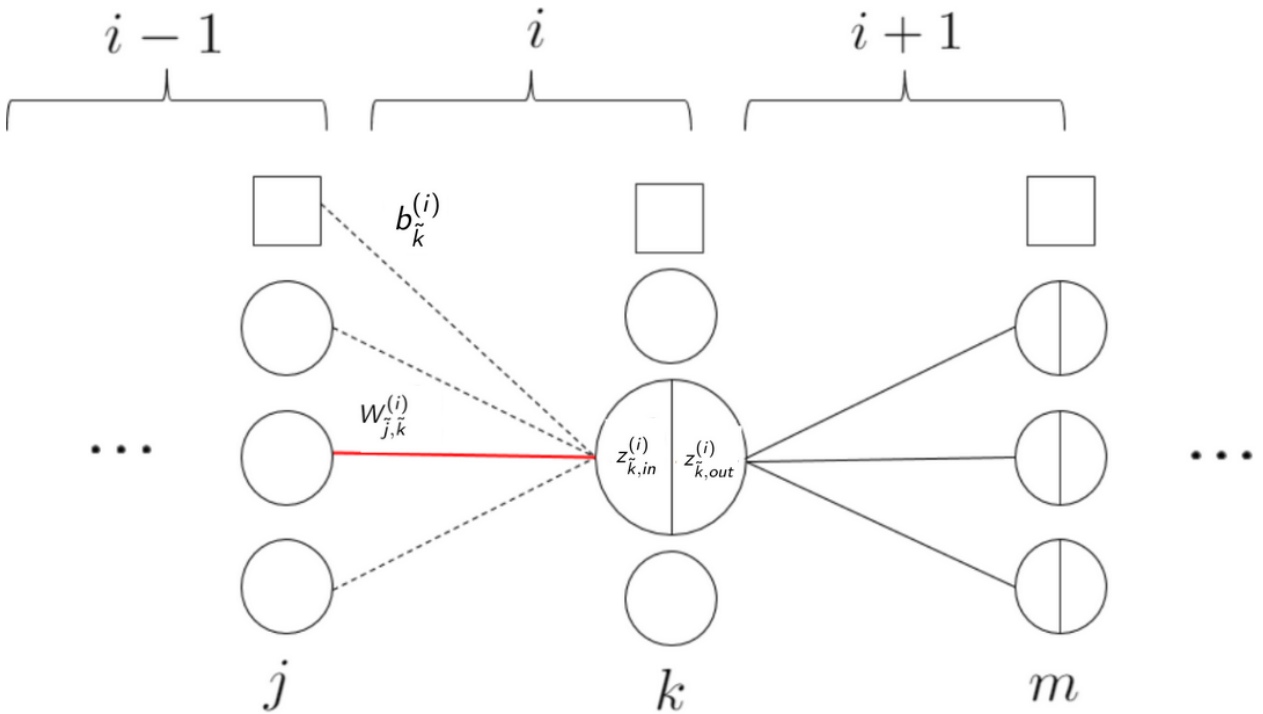
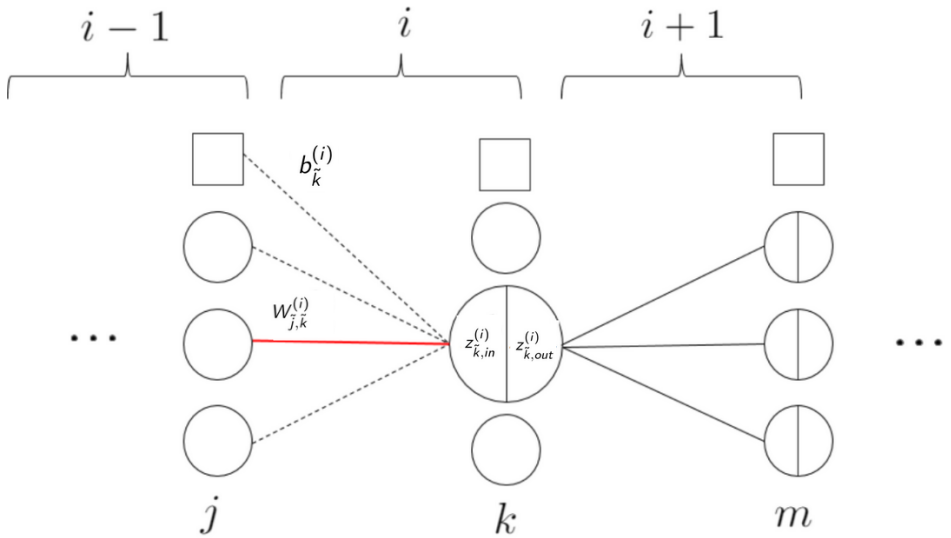


Figure: (Hallström, 2016)

BACKPROP: RECURSION



- Let $\delta_{\tilde{k}}^{(i)}$ (also: error signal) for a neuron $\tilde{k}$ in layer i represent how much the loss L changes when the input $z_{\tilde{k},in}^{(i)}$ changes:

$$\delta_{\tilde{k}}^{(i)} = \frac{\partial L}{\partial z_{\tilde{k},in}^{(i)}} = \frac{\partial L}{\partial z_{\tilde{k},out}^{(i)}} \frac{\partial z_{\tilde{k},out}^{(i)}}{\partial z_{\tilde{k},in}^{(i)}} = \sum_m \left(\frac{\partial L}{\partial z_{m,in}^{(i+1)}} \frac{\partial z_{m,in}^{(i+1)}}{\partial z_{\tilde{k},out}^{(i)}} \right) \frac{\partial z_{\tilde{k},out}^{(i)}}{\partial z_{\tilde{k},in}^{(i)}}$$

- Note: The sum in the expression above is over all the neurons in layer $i+1$. This is simply an application of the (multivariate) chain rule.

BACKPROP: RECURSION

Using

$$\begin{aligned} z^{(i)}_{\tilde{k},out} &= \sigma(z^{(i)}_{\tilde{k},in}) \\ z^{(i+1)}_{m,in} &= \sum_k W^{(i+1)}_{k,m} z^{(i)}_{k,out} + b^{(i+1)}_m \end{aligned}$$

we get:

$$\begin{aligned} \delta^{(i)}_{\tilde{k}} &= \sum_m \left(\frac{\partial L}{\partial z^{(i+1)}_{m,in}} \frac{\partial z^{(i+1)}_{m,in}}{\partial z^{(i)}_{\tilde{k},out}} \right) \frac{\partial z^{(i)}_{\tilde{k},out}}{\partial z^{(i)}_{\tilde{k},in}} \\ &= \sum_m \left(\frac{\partial L}{\partial z^{(i+1)}_{m,in}} \frac{\partial \left(\sum_k W^{(i+1)}_{k,m} z^{(i)}_{k,out} + b^{(i+1)}_m \right)}{\partial z^{(i)}_{\tilde{k},out}} \right) \frac{\partial \sigma(z^{(i)}_{\tilde{k},in})}{\partial z^{(i)}_{\tilde{k},in}} \\ &= \sum_m \left(\frac{\partial L}{\partial z^{(i+1)}_{m,in}} W^{(i+1)}_{\tilde{k},m} \right) \sigma'(z^{(i)}_{\tilde{k},in}) = \sum_m \left(\delta^{(i+1)}_{\tilde{k}} W^{(i+1)}_{\tilde{k},m} \right) \sigma'(z^{(i)}_{\tilde{k},in}) \end{aligned}$$

Therefore, we now have a **recursive definition** for the error signal of a neuron in layer i in terms of the error signals of the neurons in layer $i+1$ and, by extension, layers {i+2, i+3 . . . , O}!

BACKPROP: RECURSION

- Given the error signal $\delta_{\tilde{k}}^{(i)}$ of neuron $\tilde{k}$ in layer i, the derivative of loss L w.r.t. to the weight $W_{\tilde{j},\tilde{k}}$ is simply:

$$\frac{\partial L}{\partial W_{\tilde{j},\tilde{k}}^{(i)}} = \frac{\partial L}{\partial z_{\tilde{k},in}^{(i)}} \frac{\partial z_{\tilde{k},in}^{(i)}}{\partial W_{\tilde{j},\tilde{k}}^{(i)}} = \delta_{\tilde{k}}^{(i)} z_{\tilde{j},out}^{(i-1)}$$

  because $z_{\tilde{k},in}^{(i)} = \sum_j W_{j,\tilde{k}}^{(i)} z_{j,out}^{(i-1)} + b_{\tilde{k}}^{(i)}$

- Similarly, the derivative of loss L w.r.t. bias $b_{\tilde{k}}^{(i)}$ is:

$$\frac{\partial L}{\partial b_{\tilde{k}}^{(i)}} = \frac{\partial L}{\partial z_{\tilde{k},in}^{(i)}} \frac{\partial z_{\tilde{k},in}^{(i)}}{\partial b_{\tilde{k}}^{(i)}} = \delta_{\tilde{k}}^{(i)}$$

BACKPROP: RECURSION

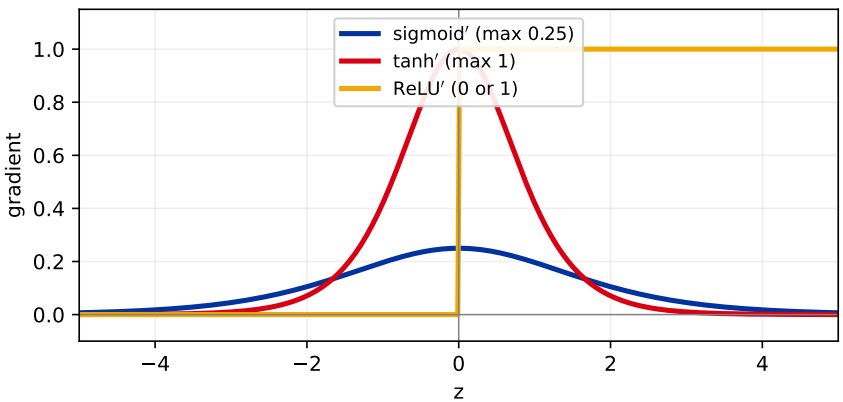
- It is not hard to show that the error signal δ^i for an entire layer i is ($\odot$ = element-wise product):
  - $\delta^{(O)} = \nabla_{f_{out}} L \odot \tau'(f_{in})$
  - $\delta^{(i)} = W^{(i+1)} \delta^{(i+1)} \odot \sigma'(z_{in}^{(i)})$
- Therefore, backpropagation works by computing and storing the error signals **backwards**. That is, starting at the output layer and ending at the first hidden layer. This way, the error signals of later layers **propagate backwards** to the earlier layers.
- The derivative of the loss L w.r.t. a given weight is computed efficiently by plugging in the cached error signals, thereby avoiding expensive and redundant computations.

REFERENCES

Hallström, E. (2016, December 4). *Backpropagation from the beginning*. Medium. https://medium.com/@erikhallstrm/backpropagation-from-the-beginning-77356edf427d

What this recursion costs: vanishing gradients

LMU's recursion $\delta^{(i)} = \mathbf{W}^{(i+1)}\delta^{(i+1)} \odot \sigma'(\mathbf{z}_{\text{in}}^{(i)})$ multiplies by σ' at *every* layer on the way back – and that factor is small:



- Sigmoid: $\sigma' \leq 0.25$. Over k layers the σ' factors alone give $\leq 0.25^k$ – unless the weights grow to compensate, the gradient **vanishes exponentially**.
- $0.25^{10} \approx 10^{-6}$: over 10 sigmoid layers those factors alone shrink it at least a *millionfold*.
- **ReLU** ($\sigma' = 1$ for $z > 0$) largely fixes this – the default hidden activation.

The **vanishing-gradient** problem is a direct consequence of the backprop recursion LMU just derived – and a big reason deep nets were hard to train before ReLU.

Deep Learning

Hardware and Software

Learning goals

- GPU training for accelerated learning
- Software for hardware support
- Deep learning software platforms

Hardware for Deep Learning

HARDWARE FOR DEEP LEARNING

- Deep neural networks require special hardware to be trained efficiently.
- The training is done using **G**raphics **P**rocessing **U**nits (GPUs) and a special programming language called CUDA.
- Training on standard CPUs takes a very long time.

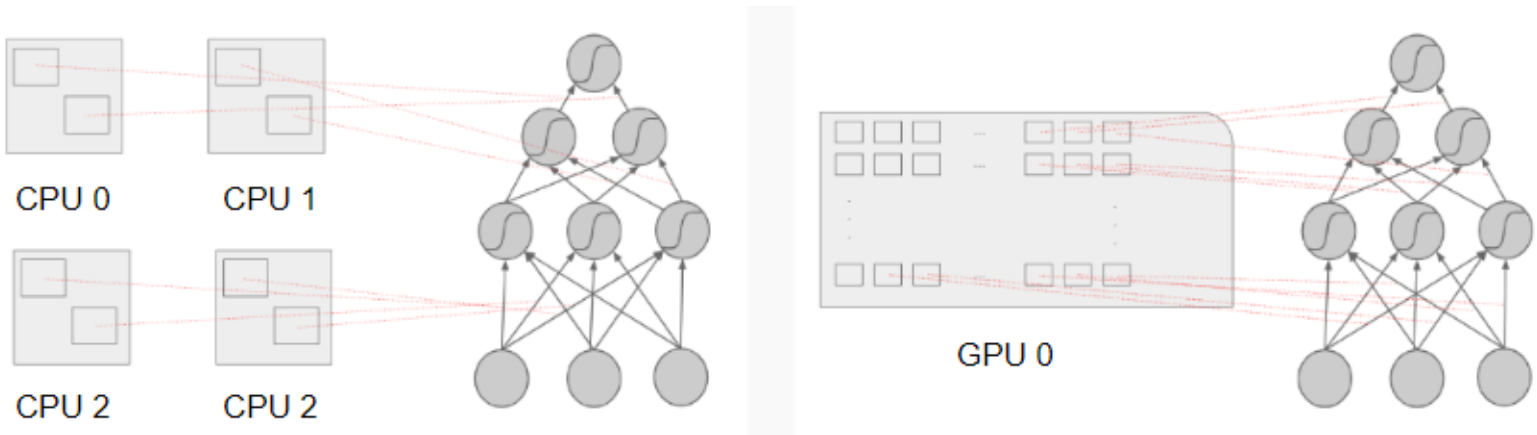


Figure: *Left:* Each CPU can do 2-8 parallel computations. *Right:* A single GPU can do thousands of simple parallel computations.

GRAPHICS PROCESSING UNITS (GPUS)

- Initially developed to accelerate the creation of graphics
- Massively parallel: identical and independent computations for every pixel
- Computer Graphics makes heavy use of linear algebra (just like neural networks)
- Less flexible than CPUs: all threads in a core concurrently execute the same instruction on different data.
- Very fast for CNNs, RNNs need more time
- Popular ones: GTX 1080 Ti, RTX 3080 / 2080 Ti, Titan RTX, Tesla V100 / A100
- Hundreds of threads per core, few thousands cores, around 10 teraFLOPS in single precision, some 10s GBs of memory
- Memory is important - some SOTA architectures do not fit GPUs with <10 GB

TENSOR PROCESSING UNITS (TPUS)

- Specialized and proprietary chip for deep learning developed by Google
- Hundreds of teraFLOPS per chip
- Can be connected together in *pods* of thousands TPUs each (result: hundreds of **peta**FLOPS per pod)
- Not a consumer product! Can be used in the Google Cloud Platform (from 1.35 USD / TPU / hour) or Google Colab (free!)
- Enables DeepMind to make impressive progress : AlphaZero for Chess became world champion after just 4 hours of training concurrently on 5064 TPUs

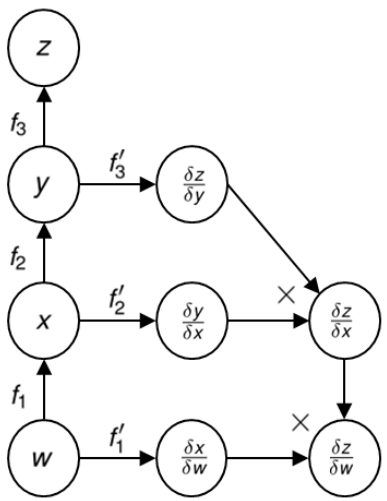
AND EVERYTHING ELSE...

- With such powerful devices, memory/disk access during training become the bottleneck
  - Nvidia DGX-1: Specialized solution with eight Tesla V100 GPUs, dual Intel Xeon, 512 GB of RAM, 4 SSD disks of 2TB each
- Specialized hardware for on-device inference
  - Example: Neural Engine on the Apple A11 (used for FaceID)
  - Keywords/buzzwords: *Edge computing* and *Federated learning*

Software for Deep Learning

SOFTWARE FOR DEEP LEARNING

- CUDA is a very *low level* programming language and thus writing code for deep learning requires a lot of work.
- Deep learning (software) frameworks:
  - Abstract the hardware (same code for CPU/GPU/TPU)
  - Automatically differentiate all computations
  - Distribute training among several hosts
  - Provide facilities for visualizing and debugging models
  - Can be used from several programming languages
  - Based on the concept of *computational graph*



SOFTWARE FOR DEEP LEARNING

Tensorflow

- Popular in the industry
- Developed by Google and open source community
- Python, R, C++ and Javascript APIs
- Distributed training on GPUs and TPUs
- Tools for visualizing neural nets, running them efficiently on phones and embedded devices.

Keras

- Intuitive, high-level **wrapper** on Tensorflow for rapid prototyping
- Python and (unofficial) R APIs

SOFTWARE FOR DEEP LEARNING

Pytorch

- Popular in academia
- Supported by Facebook
- Python and C++ APIs
- Distributed training on GPUs

MXNet

- Open-source deep learning framework written in C++ and cuda (used by Amazon for their Amazon Web Services)
- Scalable, allowing fast model training
- Supports flexible model programming and multiple languages (C++, Python, Julia, Matlab, JavaScript, Go, **R**, Scala, Perl)

Hardware: LMU's pages vs. 2026

The principle holds: training is massively parallel matrix math, so it runs on GPUs and TPUs. The numbers have moved a lot:

	on LMU's pages	**2026**
data-center GPU	Tesla V100, A100	H100 (80 GB), H200 (141 GB), B200 (192 GB); Vera Rubin announced for 2026
desktop GPU	GTX 1080 Ti, RTX 3080	RTX 5090 (32 GB, 2025)
memory per GPU	"some 10s of GBs"	80 to 192 GB on one data-center GPU
Google TPU	pods of thousands	Ironwood (7th generation, 2025): 192 GB per chip, pods of 9,216 chips
one training box	DGX-1: 8× V100	DGX B200: 8× B200

Memory, not speed, is usually what stops you. A model that does not fit in GPU memory does not train, however fast the chip. The renting section at the end of this deck does the arithmetic.

Software, and one correction

LMU's framework pages have aged too:

- **MXNet** was retired in 2023 (moved to the Apache Attic).
- **Keras 3** (2023) is no longer TensorFlow-only: it runs on JAX, TensorFlow or PyTorch.
- **PyTorch** has been governed by the independent PyTorch Foundation (part of the Linux Foundation) since 2022, not by Facebook alone.
- **AlphaZero** (Silver et al., 2017): of the 5064 TPUs, 5000 generated self-play games and 64 trained the network. After 4 hours it was rated above the *Stockfish* chess engine – it beat a program, not a world champion.

What to keep: you write the model; the framework builds the graph and runs backprop on the GPU. This course uses **PyTorch**.

The whole by-hand pass in one call: `loss.backward()`

LMU's XOR network, same weights, $\mathbf{x} = (1, 0)$, $y = 1$, in PyTorch:

```
x = torch.tensor([1., 0.]); y = 1.0
W = torch.tensor([[-0.07, 0.22], [0.94, 0.46]], requires_grad=True)
b = torch.tensor([-0.46, 0.10], requires_grad=True)
u = torch.tensor([-0.22, 0.58], requires_grad=True)
c = torch.tensor(0.78, requires_grad=True)

# forward pass
f = torch.sigmoid(u @ torch.sigmoid(W @ x + b) + c)
loss = 0.5 * (y - f) ** 2
# backward pass: every gradient at once
loss.backward()
# f = 0.7553, u.grad[0] = -0.0168, W.grad[0, 0] = 0.0023
```

Twenty pages of chain rule, one line of code. PyTorch agrees with the *corrected* forward pass ($f = 0.7553$); LMU's pages, built on 0.7525, show -0.0171 and 0.0024 for the same two gradients.

Then why learn backprop at all? It is a leaky abstraction

`loss.backward()` returns correct derivatives of whatever you wrote. Whether they are *useful* depends on what flows backward – "backpropagation is a leaky abstraction" (Karpathy, 2016):

- **Saturation:** in the flat tails of sigmoid and tanh the local derivative is ≈ 0, so the gradient stops there (the vanishing-gradient frame).
- **Dead ReLUs:** a unit that is negative for every input gets zero gradient forever – from initialization, or after a too-large update knocks it off the data.
- **Exploding gradients:** multiply by the same weight matrix step after step, as a recurrent net does, and the gradient can blow up (RNN chapter).
- **A real bug:** a deep Q-learning implementation clipped its error term to tame outliers. Clipping has local derivative 0 outside its range, so exactly those examples got *zero* gradient and were silently ignored. The fix: the Huber loss.

Autograd never complains: the code runs and the gradients are "correct". Knowing the backward pass is how you notice they are useless.

Renting a GPU

Your laptop trains the homework. Anything bigger runs on someone else's GPU, by the hour.

Where to get a GPU

option	what you get	cost	the catch
Google Colab, free	a T4 (16 GB) when one is available	free	no guaranteed GPU; sessions disconnect
Colab Pro	100 compute units a month; a T4 uses about 2 an hour; A100 when available	\$9.99 / month	units run out; GPU type not guaranteed
Kaggle notebooks	about 30 GPU-hours a week (P100 or 2× T4)	free	12 h per session
GPU clouds (e.g. RunPod, Lambda)	any GPU, full control: RTX 4090 from \$0.34/h, A100 80 GB from \$1.19/h, H100 \$1.99 to \$3.29/h	per second or hour	you pay while it is *on*, idle or not
AWS, Google Cloud, Azure	everything, integrated with storage and data	highest on demand	spot instances are cheaper but can be taken away mid-run

A sense of scale: 10 hours on one H100 at \$1.99/h is about \$20. Whether your model even fits on it is the next slide.

Prices checked 17 September 2026 (RunPod community cloud, Lambda on-demand, Colab and Kaggle plan pages); they change every few months.

Will it fit? Memory before you rent

Count **bytes per weight**. A full-precision number takes 4 bytes, a half-precision one 2. Training with Adam in mixed precision needs about **16 bytes per weight** (Rajbhandari et al., 2020):

$$\underbrace{2}_{\text{fp16 weights}} + \underbrace{2}_{\text{fp16 gradients}} + \underbrace{4+4+4}_{\text{fp32 weights, Adam's } m,\ v} = 16$$

model	**weights**	**run (2 B/weight)**	**train (16 B/weight)**
homework MLP	0.24M	0.5 MB	4 MB
GPT-2 small	124M	0.25 GB	2 GB
GPT-2 XL	1.5B	3 GB	24 GB
7B language model	7B	14 GB	**112 GB**

Activations come *on top* and grow with the batch size. So one 80 GB H100 trains at most about a 5B-parameter model the plain way; the 7B model does not fit. Adam's two moment vectors alone are 8 of the 16 bytes – the optimization deck shows where they come from.

Renting without getting burned

- **The meter runs while the machine is on,** busy or idle. Stop it when you are done; stored disks are often still billed after that.
- **The disk can vanish.** A Colab runtime or a cloud container is wiped when it stops. Save checkpoints and results somewhere persistent, often.
- **Expect interruptions.** Colab disconnects, spot instances get reclaimed: checkpoint every few epochs so a restart resumes instead of starting over.
- **Check you actually got the GPU:** `nvidia-smi` in a terminal, then `device = "cuda" if torch.cuda.is_available() else "cpu"`, and move *both* the model and every batch with `.to(device)`.
- **Keys never go in a notebook.** API keys (W&B, Hugging Face) belong in environment variables or the platform's secrets manager: notebooks get shared and screenshotted.

Debug where it is free. Run one epoch on a tiny subset on your laptop or a free notebook first; rent the big GPU only for a run you already trust.

Tracking experiments

Twenty runs in, nobody remembers which curve came from which settings.

What an experiment tracker records

Homework task 3 alone is three learning rates. Add dropout, weight decay and BatchNorm and you have dozens of runs. A tracker writes the lab notebook for you, for *every* run:

- **Config:** every hyperparameter the run used.
- **Metrics over time:** train and validation loss per epoch, accuracy, anything you log.
- **System stats:** GPU memory and utilization – is the rented GPU even busy?
- **Code and environment:** the git commit and the installed package versions.
- **Artifacts:** checkpoints, sample predictions, tables.

The tool matters less than the discipline: **no experiment without its settings recorded**. By hand, that habit dies around run three.

Weights & Biases in a training loop

Weights & Biases (W&B): create a free account, run `wandb login` once, then:

```
import wandb

config = dict(lr=1e-3, hidden=256, dropout=0.2, epochs=30)

with wandb.init(project="fashion-mlp", config=config) as run:
    model = MLP(config["hidden"], config["dropout"]) # your model
    opt = torch.optim.AdamW(model.parameters(), lr=config["lr"])
    for epoch in range(config["epochs"]):
        train_loss = train_one_epoch(model, opt) # your loop
        val_loss, val_acc = evaluate(model)
        run.log({"train_loss": train_loss, "val_loss": val_loss,
                 "val_acc": val_acc, "epoch": epoch})
```

Every run gets a page with live charts. No internet on the machine? Set `WANDB_MODE=offline` and upload later with `wandb sync`.

What the dashboard gives you

- **Overlaid curves:** all runs on one chart, grouped or colored by a config value – the regularization deck's "read your curves", for twenty runs at once.
- **A sortable runs table:** sort by best validation loss; the winner's config is one click away.
- **Sweeps:** W&B can run the grid, random or Bayesian search from the tuning lecture for you.
- **Cost:** free for personal use; a free Pro license for academic accounts.
- **Alternatives:** TensorBoard (local, free; PyTorch has a built-in writer) and MLflow (open source, self-hosted).

Try it on the homework: log each learning rate in task 3 as its own run and compare them on one chart. (W&B has been part of CoreWeave, a GPU cloud, since 2025 – renting and tracking now come from the same company.)

Putting it together

One objective, one backward sweep, and the hardware that makes it fast.

Recap: how a network learns

- **Objective:** minimise the empirical risk by **gradient descent**; **SGD** does it on mini-batches – cheaper, noisier, and often generalises better.
- **Backprop** = the chain rule on the computational graph, run *backward once*. Each layer's error signal δ is computed from the layer above and reused: "$\delta\times$ local derivative", "$\delta\times$ input".
- That reuse is why *one* backward pass gives **all** gradients – the naive alternative is one forward pass per weight.
- **You won't code it by hand:** frameworks (PyTorch, TensorFlow) build the graph and run backprop when you call `loss.backward()` – on **GPUs/TPUs** that do the matrix math massively in parallel, rented by the hour and with every run tracked.

We can now *train* a network. What remains: keeping it from **overfitting** (regularization), and **optimizing** faster than plain SGD – the next two chunks.